DOCUMENT RESUME

ED 028 408 AL 001 357

By-Gross, Louis N.

A Computer Program for Testing Grammars On-Line.

Pub Date Jul 68
Note-65p.
EDRS Price MF-$0.50 HC-$3.35
Descriptors-*Computational Linguistics, Computer Assisted Instruction, Computer Programs, *Tests, *Transformation Generative Grammar
Identifiers-Compatible Time Sharing System, CTSS, LISP

This paper describes a computer system which is intended to aid the linguist in building a transformational grammar. The program operates as a rule tester, performing three services for the user through sets of functions which allow the user to--specify, change, and print base trees (to which transformations would apply); define transformations and specify the conventions for applying them; and apply these transformations to the base trees. It is expected that the rule tester would be of use to a linguist who has written a transformational grammar in which his rules were explicit. He would use the rule tester to find out whether his rules actually do what he thinks they do, to modify the grammar when necessary, and to observe the workings of his rules in detail. The rule tester might be of use in teaching an introductory course in transformational grammar, enabling the student to observe the workings of the machinery of transformational application and to see his errors pointed out to him immediately. The program described here runs on the M.I.T. Compatible Time Sharing System and uses the LISP programming language. While the description of the program given in this paper is not sufficiently detailed to enable the linguist with no experience with computers and the CTSS system to use the program, it is hoped that the linguist will be able to judge whether it would be useful for him. A model "script" is appended. (JD)

ED028408

U.S. DEPARTMENT OF HEALTH, EDUCATION & WELFARE
OFFICE OF EDUCATION

THIS DOCUMENT HAS BEEN REPRODUCED EXACTLY AS RECEIVED FROM THE PERSON OR ORGANIZATION ORIGINATING IT. POINTS OF VIEW OR OPINIONS STATED DO NOT NECESSARILY REPRESENT OFFICIAL OFFICE OF EDUCATION POSITION OR POLICY.

A COMPUTER PROGRAM FOR TESTING GRAMMARS ON-LINE

by

Louis N. Gross
The MITRE Corporation
Bedford, Massachusetts

and

Massachusetts Institute of Technology
Cambridge, Massachusetts

July 1968

AL 001 357

Table of Contents

	Page
INTRODUCTION	1
What the Program Does	2
Further Development of the Rule Tester	3
Advice to the Reader	4
Acknowledgements	5
THE RULE TESTER	
Some Generally Useful Functions	7
Functions for General Tree Management	8
Functions Which Modify Trees Printed by PRTREE	11
Functions That Do Things with Transformations	17
Erasing	22
Pruning	23
Functions for applying transformations	25
REFERENCES	30
APPENDIX: A SCRIPT WITH COMMENTS	31

INTRODUCTION

This paper contains a description of a computer program which is intended to aid the linguist in building a transformational grammar. The behavior of the program might be compared to that of a literal-minded, but indefatigable assistant who has been taught the formalism of transformational grammar, but has no 'feel' for what it should do. This assistant will do a very good job of checking the mechanical operation of the transformations against pre-defined trees, help with the bookkeeping of changes to the grammar and to a set of test trees, but he will not come up with original ideas or use his intuition. Because of these characteristics, the linguist must be explicit in writing his grammar, a feature which has obvious theoretical advantages.

The program described here runs on the M.I.T. Compatible Time Sharing System (C.T.S.S); it is designed for the linguist to use "interactively". That is, the use of the system is in the nature of a conversation between the linguist and the program, the communication taking place by means of a typewriter or teletype.

Since the main reason for the existence of the program is its use as a tool by the linguist, this paper is being distributed to find out whether the program actually meets the needs of working linguists. While the description is not sufficiently detailed to enable the linguist who has no experience with computers and the CTSS system to use the program, it is hoped that it is sufficient to enable the linguist to judge whether it would be useful for him. There is an appendix consisting of a "script" of a console session in which I use the program to test the operation of and modify a sample grammar.

If only a few people are interested in using the program, then several console sessions would probably serve to acquaint them with it and with the time-sharing system. If a substantial number of people use the program, then this fact will serve as an incentive for me to write a more detailed manual which would allow someone without prior experience with computers to use the program with very little extra help.

What the Program Does

There are three services that the program will perform for the user:

1. There is a set of functions that allow the user to specify, change, and print (in a "natural" form) a tree. Typically, this set would be used to construct the base trees to which the transformations would apply.

2. There is a set of functions allowing the user to define transformations and specify the conventions for applying them (e.g., order, optionality).

3. There is a set of functions for applying the transformations defined by the functions in 2 to one of the trees specified by those in 1. The user (if he wishes) can see the result of applying each transformation as it applies. If a transformation that can apply is optional, the user is asked whether he wants it to apply.

It is expected that the rule tester would be of use to a linguist who has written (or would write if he had such a device to test it) a transformational grammar (or fragment thereof) in which his rules were explicit. He could use the rule tester for the following purposes:

1. To find out whether his rules actually do what he thinks they do (i.e., whether they actually do map the base trees of his base component into what he thinks are the corresponding surface trees), and to modify the grammar when it turns out that they do not.

2. To observe the workings of his rules in detail. The rule tester might be of use in teaching an introductory course in transformational grammar. A student using this system could appreciate the notions "transformation," "deep structure," "structural description," etc., by observing the workings of the machinery of transformational application without having to do the tedious pencil work otherwise required. Also, the student would benefit from having the errors that he inevitably will make pointed out to him immediately.

Future Development of the Rule Tester

Future development of the rule tester depends crucially on the users of the system. If no one uses it, there is no point in doing any further work on it. If a user appears who would like the system to do something which it does not do now, I will attempt to modify the program so that it does.

This paper is already obsolete in that syntactic features were added to the program after the body of the paper was written. However, since the paper as presented here does describe the general performance of the program, it does not seem to be worth updating the manual unless it is first determined that the program will be used. A sketchy description of the addition is presented in the next paragraph.

It is now possible to specify, for each node in a tree, a list of inherent feature names and values, and to test (in determining whether

a transformation applies to a tree) whether a node matching a segment of a structural description has certain features. Contextual features have not been added since they do not (with the usual exception of Walbiri) seem to have any bearing on the application of transformations. If anyone wishes to study lexical insertion, however, it would be possible to add contextual features. Further improvements can be made fairly easily. For example, the notion of "command" has not been programmed, but it could be if someone were to be interested in using the rule tester to study this phenomenon.

Advice to the Reader

Familiarity with the use of CTSS (the M.I.T. time sharing system) and with the LISP programming language is assumed in the sense that many details will not be clear if the reader lacks this familiarity. The reader is advised to read through the paper quickly, start going through the "script" in the Appendix and then refer back to the paper (or contact the author) if it is not clear what the program is doing.

Most of the time the user communicates with the system by typing directly to the LISP interpreter. This interpreter expects to receive pairs which consist of a function name followed by a list of arguments enclosed in parentheses. The system pays attention to the user's input only when he gives it a carriage return after he has typed in at least one pair consisting of a function name and a list of arguments (the parentheses for which must balance). The user may, however, type several function-argument list pairs on one line and have LISP process them sequentially after he has given a carriage return.

The typewriter input program allows for the user's commission and correction of typing errors by using the character # to ignore the previous character (a space counts as a character), and @ to ignore all the previous characters on the line being typed. Thus,

thwisiswiskwis@thui@thu#is isa##s a lint#e #without ewwo###rrors

is processed by the program as if it had been typed:

this is a line without errors

Acknowledgements

The predecessor to the program described in this paper was implemented at MITRE on an IBM 7030 computer using DD-13 display consoles with vector generating capabilities and lightgun control (Gross, 1967). Thus it was possible to display trees in their more familiar format directly on the face of the scope and to manipulate them using the lightgun and lightbutton marginal control words. Trees could be generated by cycling through the phrase structure rules, displaying alternative expansions on the display scope, and selecting the desired one. Transformational rules could be applied to a base tree to produce the corresponding surface tree. Correspondingly, the deep structure of a sentence could be produced, following the logic of the MITRE Syntactic Analysis Procedure (Zwicky, et al., 1965; Walker, et al., 1966), in which presumable surface trees for the sentence are determined first, and then reversal rules (inverse transformations) are applied to yield its base tree.

Several other rule testers have been written, and the program described in this paper incorporates many ideas from these other systems. Joyce Friedman (1966-1968) has programmed an off-line rule

tester (in FORTRAN) at the Department of Computer Sciences, Stanford University, and is using it to test transformational grammars from directed random generation of sentences through application of transformations to lexical insertion.

Blair (1966) has written an off-line system (in LISP) at the IBM Watson Research Center for testing and updating a transformational grammar.

Londe and Schoene (1967) have written an on-line interactive rule tester using a display console in conjunction with a teletype.

I would also like to express my gratitude to people who have contributed grammars. Playing with their grammars has resulted in a number of improvements in the program. Dave Vetter wrote the grammar from which the example in the Appendix takes off. Hugh Matthews contributed a grammar of Hidatsa, and discussions with him resulted in a number of improvements. Susumo Kuno contributed a set of transformations for Japanese. Jay Keyser provided a version of Rosenbaum's grammar for English complement constructions.

THE RULE TESTER

Some Generally Useful Functions

The functions described in this subsection, while not relating to any specific aspect of the rule tester, are of general use. They are described here since they may be used in conjunction with any part of the rule tester.

CSET(symbol value) [set symbol to value]

causes the symbol 'symbol' to have the value 'value'.

LOADG((file1, ... ,filen)) [load files]

cauzes the n files whose names are 'file1' LISP, ... , 'filen' LISP to be loaded into the system, printing the first name of each file just before loading it. If all the files are successfully loaded, NIL is then typed.

DEFFILE(filename listofthings) [define from file]

'filename' is the first name of a file 'filename' LISP, and 'listofthings' is a list (surrounded by parentheses) of names of functions, symbols, or transformations which are defined on that file. Causes only those symbols, transformations, or functions whose names are in 'listofthings' to be read in from 'filename' LISP, after which a list of the names of things which could not be found (NIL if all were found) is printed.

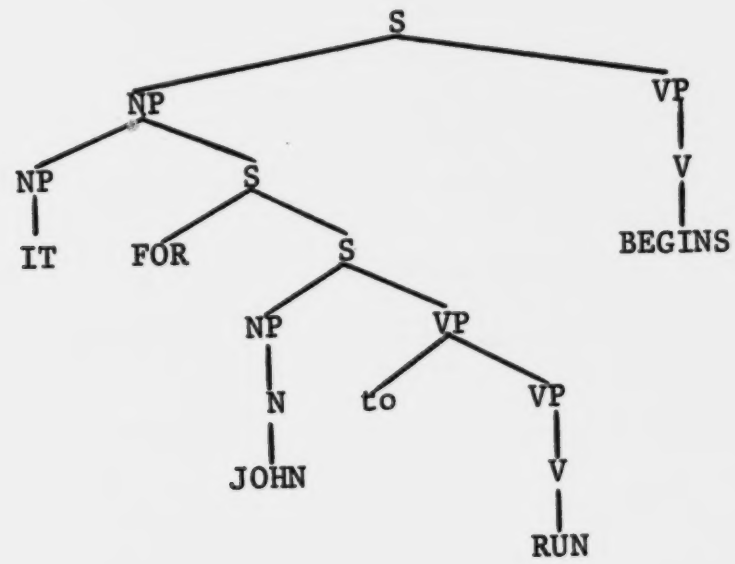
Functions for General Tree Management

TREESET(loc tree) [set tree]

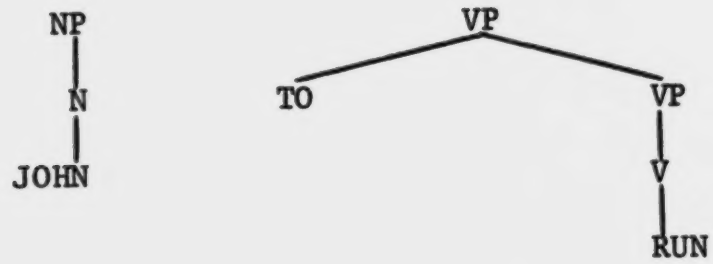
'loc' is a symbol and 'tree' is a tree in a list form to be described by example, or the name of such a tree.

The following statement is used to set 'TRANTREE' to the tree below it:

```
TREESET(TRANTREE
(S(NP(NP(IT) S(FOR S(NP(N(JOHN))VP(TO VP(V(RUN))))))
VP(V(BEGINS))))     )
```



The list format for inputting trees could be described as follows: a (multiply rooted) tree is a list where each symbol on the top level of the list (i.e., not enclosed in parentheses) is the name of a node at the top of the tree. If any node has daughters, then the daughter tree is a multiply rooted tree in the same form as this definition of a tree. The usual referent for the term tree is a special case in which there is only one node on the top level of the list. However, in the analysis portion of the program, it is useful to consider as a tree, e.g., the following subtree of the above tree:



(NP (N (JOHN))VP (TO VP(V(RUN))))

PRTTREE(tree) [print tree]

'tree' is a tree or the name of a tree. The tree whose name is 'tree' is printed out in a 'sideways' format that could be described as a conventional tree which has been rotated 90° so that its top node(s) are at the left, has been flipped over with each set of daughters reordered so that leftmost daughters in the original tree correspond to topmost daughters in the new tree (the concept of 'firstness' being preserved), has been squashed so that a node and all of its first-born descendants appear on the same line, and then has had its lines removed. Each node is also preceded by a unique number (which may be used to refer to that node later), and when a node which appears in the rightmost position on the page has daughters, '...' is printed after it (to indicate that there is some tree at that point which is off the page). For example, if 'TRANTREE' has been set as described in TREESET, then

PRTREE(TRANTREE)

produces the following printout:

```
1S          2NP          3NP          4IT
                         5S           6FOR
                                      7S          8NP         9N          10JOHN
                                                  11VP        12TO
                                                              13VP        14V ...
            16VP         17V          18BEGINS
```

18

the value of PRTREE (18 at the lower left) is the largest node number reached (including those which could not be printed on the page). The system variable MPLEV specifies the number of tab's (i.e., one less than the number of nodes) printed on a line. For this example,

CSET(MPLEV 6)

has been executed. It is up to the user to set the tabs on his typewriter appropriately. The timing used by the time sharing system (i.e., the time it waits for a carriage to return before typing the next line) is based on an assumption of tabs set to 11,21,31,... In addition, LSTPTREE is set to point to 'tree'.

PNODE(node) [print node]

'node' is the number or symbol of any node (including ones off the page) in the most recently printed tree. The tree (including its sisters) dominated by the node whose number is 'node' is printed, with the numbering beginning with the number of that node. If 'node' is a symbol, then the first (lowest numbered) node in the tree with that symbol will be printed. For example, PNODE(8) after the tree above has been printed produces the printout:

```
8NP        9N          10JOHN
11VP       12TO
           13VP        14V         15RUNS
```

PLEAFS(tree) [print leaves]

'tree' is a tree or the name of a tree the list of leaves (i.e., terminal nodes) of 'tree' is printed out. For example,

```
PLEAFS(TRANTREE)
```

prints

```
(IT FOR JOHN TO RUN BEGINS)
```

INTREE(tnam tree) [input tree]

sets 'tnam' to the tree 'tree' (expressed as a list in the format described for TREESET) and prints it with PRTREE.

FILETREE(treename filename) [add tree to file]

changes the file 'filename' LISP so that it has appended to the end of it a function which will, when the file 'filename' is read in (e.g., by LOADG), cause 'treename' to be defined as the same tree it was when this FILETREE was typed.

Functions Which Modify Trees Printed by PRTREE.

The following functions may be used to modify the tree most recently printed by PRTREE. Each function has two arguments: 'place', which indicates the place in the tree to apply the modification; and 'newtree' (except for CHVAL) which indicates the tree to be put there. If 'place'

is an integer, then it is assumed to refer to the node whose number would be 'place' if the tree were to be printed in its current state. This latter number may differ from the number printed in the last print if a change was made to the tree at any node with a number lower than 'place'. If 'place' is a symbol, then it refers to the first node in the tree whose name is 'place'. If 'newtree' is a positive integer, then it refers to a subtree of the tree in the same way as does 'place' when it is an integer. The tree being adjoined at 'place' is a copy of the tree beginning as the node addressed, and it includes any sisters of the tree. If 'newtree' is a negative integer then the tree being adjoined at 'place' does not include any sisters of the tree addressed by 'newtree'. If 'newtree' is a symbol, then it is the name of a tree already defined. Otherwise it may be a tree in the format required by TREESET or INTREE.

CHDAT(place newtree) [change left daughter]

changes the leftmost daughter of 'place' so that it is 'newtree'.

CHSIS(place newtree) [change right sister]

changes the right sister of 'place' so that it is 'newtree'.

CHVAL(place symbol) [change value (node name)]

changes the node name (value) of 'place' so that it is the symbol 'symbol' (havoc will probably be wrought if 'symbol' is not a symbol).

CHNODE(place newtree) [change node]

changes the node name of 'place' to that of 'newtree' and the daughter of 'place' to the daughter of 'newtree' (i.e., it changes the entire subtree dominated by the node at 'place' so that it is the subtree dominated by the node 'newtree').

IDAT(place newtree) [insert daughter]

changes 'place' so that its leftmost daughter is 'newtree' and the rightmost sister of the new daughter is the old leftmost daughter of 'place'.

ISIS(place newtree) [insert sister]

changes 'place' so that 'newtree' is inserted between it and its current right sister.

The use of the above routines to input and modify a tree is illustrated on the following pages. Note that the lines preceded by a '*' were typed by the user; the other lines were typed by the program.

```
*  INTREE(TEST (S(A B(C D)E(F(G(H(I J)))K)))
*  )
    1S          2A
                3B          4C
                            5D
                6E          7F          8G          9H          10I
                                                                11J
                            12K
    12
*   CHVAL(K L)
    DONE
*  PNODE(7)
    7F          8G          9H          10I
                                        11J
    12L
*  CHDAT(3 (M N(O)P))
    DONE
*  PNODE(3)
    3B          4M
                5N          60
                7P
    8E          9F          10G         11H         12I
                                                    13J
                14L
*  IDAT(5 10)  PNODE(4)
    DONE
    4M
    5N          6G          7H          8I
                                        9J
                100
    11P
*  CHDAT(6 -8)  PNODE(5)
    DONE
    5N          6G          7I
                80
    9P
*  PNODE(1)
    1S          2A
                3B          4M
                            5N          6G          7I
                                        80
                            9P
                10E         11F         12G         13H         14I
                                                                15J
    -                       16L
```

```
*  CHDAT(12 14)  PNODE(F)
    DONE
    11F          12G          13I
                              14J
    15L
*  CHSIS(4 (R(S T)U))   PNODE(1)
    DONE
    1S           2A
                 3B           4M
                              5R           6S
                                           7T
                              8U
                 9E           10F          11G          12I
                                                        13J
                              14L
*  ISIS(R 10)   PNODE(4)
    DONE
    4M
    5R           6S
                 7T
    8F           9G           10I
                              11J
    12L
    13U
*  PND#ODE(1)
    1S           2A
                 3B           4M
                              5R           6S
                                           7T
                              8F           9G           10I
                                                        11J
                              12L
                              13U
                 14E          15F          16G          17I
                                                        18J
                              19L
*  ISIS(F -10)   PNODE(1)
    DONE
    1S           2A
                 3B           4M
                              5R           6S
                                           7T
                              8F           9G           10I
                                                        11J
                              12I
                              13L
                              14U
                 15E          16F          17G          18I
                                                        19J
                              20L
```

```
* CHSIS(M -16) PNODE(3)
   DONE
   3B          4M
               5F          6G          7I
                                       8J
   9E          10F         11G         12I
                                       13J
               14L


* CHDAT(10 NIL) PNODE(10)
   DONE
   10F
   11L
* CHD#SIS(4 NIL)   PNODE(1)
   DONE
   1S          2A
               3B          4M
               5E          6F
                           7L
* INTREE(TEST1 (S(X(Y Z))))
   1S          2X          3Y
                           4Z
   4
* ISIS(3 TEST) PNODE(1)
   DONE
   1S          2X          3Y
                           4S          5A
                                       6B          7M
                                       8E          9F
                                                   10L
                           11Z
* IDAT(X -8) PNODE(1)
   DONE
   1S          2X          3E          4F
                                       5L
                           6Y
                           7S          8A
                                       9B          10M
                                       11E         12F
                                                   13L
                           14Z
* CHSIS(2 -10)   CHSIS(3 NIL) CHDAT(3 5) PNODE(1)
   DONE
   DONE
   DONE
   1S          2X          3E          4L
               5M
```

Functions That Do Things with Transformations

DEFTRAN(listofarguments) [define transformation]

This is the function used to define a transformation. It has one argument, which is a list. The explanation is most easily done in terms of examples. The Appendix begins with a list of the transformations now in the system. These transformations are Rossian in flavor, but over-simplified. Therefore, they will serve admirably for explicative purposes.

A transformation can be regarded as having five parts -- name, options, structural description (SD), restrictions, and structural change (SC). Only the name and SD are required, though it would make no sense to not have the SC (since the transformation would not do anything if it were absent so, in that case, the entire transformation might as well be absent). The structure of the five parts will be described briefly, with references to the example transformations.

1. The name of the transformation is any legal symbol, and the first item in the list of arguments is assumed to be the name (great havoc will be wrought if it isn't). For example, the names of the first two transformations are COMPLA and COMPLB.

2. All symbols after the name and before the first list are taken to be options. In general, options specify how the program handling the transformation is to be applied. For each option there is a default decision, which is what you get if the option is not specified. The default options are set by giving them as arguments to the function TRANOPT. In the current system, there are four options:

a) optional/obligatory--optional if OPT is specified, obligatory if NOPT is specified.

b) recursive/nonrecursive--recursive (applies to its own output) if RECURS is specified, non-recursive if NRECURS is specified.

c) iterative/noniterative--iterative if ITER is specified, non-iterative in NITER is specified. Iterative transformations have the SD matched in as many ways as possible and then have the SC applied to the tree for each such way.

d) bounded/not bounded--bounded if BND is specified, not bounded if NBND is specified. If a transformation is bounded, then the SD search will not go below any node whose symbol is "S" and which does not match a symbol in the SD.

The TRANOPT statement, TRANOPT((NOPT NRECURS ITER NBND)), before the transformations of the Appendix specifies that, unless a contrary option is mentioned in a transformation, they are to be obligatory, nonrecursive, iterative, and not bounded.

3. The SD begins with the first list in the list of arguments. If this list is followed by another list, then that list is also part of the SD. The second list is used to indicate those elements of the SD which themselves have SD's. Each element of the SD is given a number to be used in the SC. The penciled-in numbers in the Appendix are the numbers which are assigned by the program. Note that the SD of the transform ITREPL is traditionally written as

```
X [IT [FOR [NP  VP]]] X
  NP  S    S
1 2 3 4 5  6 7  8     9
```

The program which handles the SD sets up an array such that the ith item in the array is a pointer to the node in the tree (first and last node in the tree for variables) which matches the element of the SD numbered i. For example, the 9 elements of the SD of ITREPL match the following nodes (respectively) in the tree of Figure 1:

(NIL NIL)2 4 5 6 7 8 11 16 (NIL NIL)

In general, a SD is defined (recursively) as a list of segments, where each segment is either:

a) an X corresponding to one of the variables X,Y,,..used by linguists (except that here all variables are called X).

b) another symbol (meaning that a node whose name is this symbol must match this segment).

c) a list of the form

(* ...)

where '...' stands for a list of symbols. The segment of the SD must match a node whose name is one of the symbols on the list.

d) a list of the form

(n ...)

where '...' stands for a list of symbols and 'n' is an integer. This element is optional, and the number 'n' indicates (in case there are several optional elements in the transformation) the priority assigned to the element. ('n' should always be less than the number of optional elements--right now it can't be more than 3). The SD is first tried with all optional elements included. If there is no match, then it is tried with the optional elements deleted, deleting the highest numbered ones first, and trying to match the longest SD possible.

e) a list of the form

($ ...)

where '...' stands for a list of symbols. This element is optional, but unlike d) it is really a way of specifying the contents of a variable. When a segment of this type is the current segment, it matches the first node it finds which is on the list "...". However, if it can find none, then it matches NIL and the node at which this segment started searching is used as the starting point for the next segment.

f) an SD defined recursively by the very definition you are now reading, whose left parenthesis is labeled either by a single symbol or by a list of symbols (latter option not illustrated). If this is used then the SD actually consists of two lists, the second containing the labels for the left brackets (in the order that the left brackets appear) in the first list. Each bracket label is either a symbol or a list of symbols.

4. The restrictions specify conditions which must be met by nodes matching segments of the SD. For each segment which has restrictions, the user must provide the number of the segment followed by an expression made up of elementary restriction pairs and the Boolean operators 'AND' 'OR' 'NOT' used in the conventional way for Boolean expressions. The following two elementary restrictions are defined in the system so far:

EQA(n) [equality test]

where 'n' is the number of a segment which precedes this segment. This function is true if the tree dominated by the node matching the segment

has the same structure as the tree dominated by the node matching the 'n'th segment.

IANA(list) [immediate analysis test]
where 'list' is a list of symbols. This function is true if the list of names of the nodes of the daughters of the node matching the segment is the same as the list of symbols in 'list'.

In the example given in the Appendix, only EQA is used as a restriction, and it is not used in Boolean combination. An example of what might be used as a restriction would be something like:

3 EQA(1) OR (IANA((NP VP)) OR IANA((NP VP ADV)))

4 NOT(EQA(3) OR NOT(IANA((V NP)) AND EQA(2)))

5. The SC begins with the first element after the restrictions (if any) and is a sequence of pairs, each pair being a symbol specifying the type of change, followed by a list of arguments specifying where and (in most cases) what. Each change type is actually a LISP function, so that it would be easy to add more changes. Right now, the following are defined (and illustrated in the transformations in Appendix A):

ARS(n place) [add 'n' as a right sister to 'place']

ARD(n place) [add 'n' as a right daughter to 'place']

ALS(n place) [add 'n' as a left sister to 'place']

ALD(n place) [add 'n' as a left daughter to 'place']

SB(n place) [substitute 'n' for 'place']

CADJR(n place) [Chomsky adjoin (node raise) 'n' to the right of 'place]

CADJL(n place) [Chomsky adjoin (node raise) 'n' to the left of 'place']

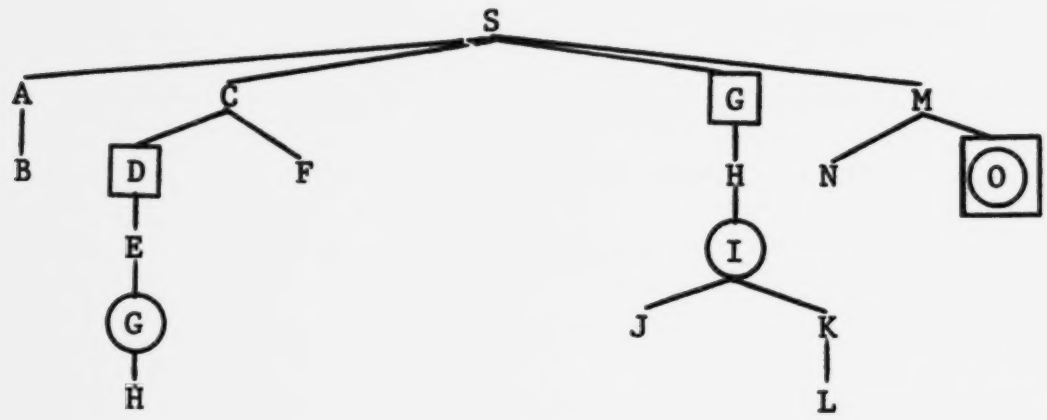
ERASN((n1 n2 ...)) [erase each node n1, n2, ... from the tree]

For each of the above functions, 'n' is either the number of a non-variable segment in the SD or a symbol, 'place' is the number of a SD segment which may match a variable except in the case of SB where it may not match a variable. If 'place' matches a variable, then functions which do things to the right get the right part of the variable, which functions which do things to the left get the left part of the variable. If 'place' does not match any node (i.e., is NIL), then the change does not happen. None of the arguments to ERASN may match variables. If 'n' is a number with a minus in front of it, then the node matched by the nth segment is erased from the tree.

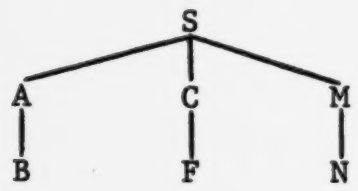
Note that the SC functions apply in the order they are written in the transformation, so that a SC function may well change the node corresponding to an 'n' in a later function (as it does in the ITREPL transformation for example).

Erasing

When a node is erased (either by a SC specification of ERASN or by SB with NIL as the first argument and that node as the second argument), then what is actually erased is the first node dominating it whose parent branches or the node itself if its parent branches. Erasing of this node consists in detaching it (and any structure it dominates) from the tree. For example, if the circled nodes in the tree below are erased



then the nodes being detached from the tree are those surrounded by squares, resulting in the following tree:



Pruning

Pruning a node involves removing it from the tree, but retaining any structure it dominates. A node is checked for prunability when one of its daughters has either been erased, substituted for, or pruned. It will be pruned if any of the following three conditions are true:

1. The node immediately dominates only one node which has the same name, e.g., a structure of the form:

2. The node does not branch and its name is on the list MUSTBRANCH. Thus the user can indicate which nodes are to be pruned

if they do not branch by setting MUSTBRANCH to a list of those names.

3. The user has indicated the name of this node as necessarily immediately dominating a node which it does not immediately dominate. The user indicates this by setting MUSTDOM to a list of pairs where the first element of each pair is a node name and the second is a list of node names one of which must be immediately dominated by the node name in the first element. For example, if the user has executed

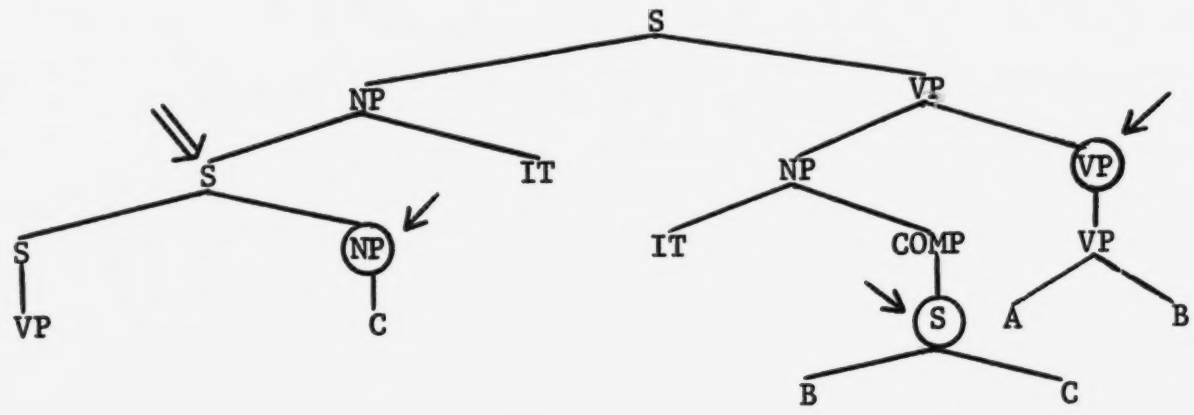
CSET(MUSTDOM(S(VP) NP(N IT)))

then any S node which does not immediately dominate a VP, or any NP node which immediately dominates neither a N or IT will be pruned.

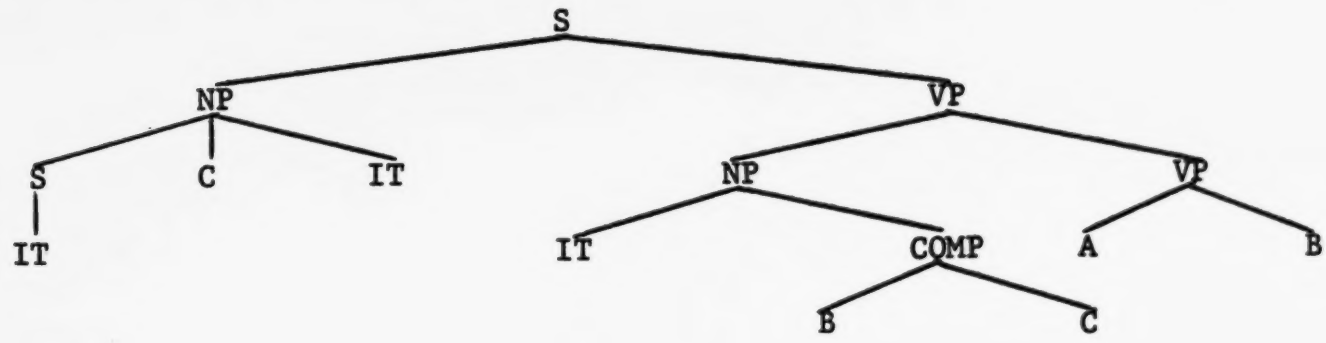
Example: suppose the user has set MUSTDOM and MUSTBRANCH as follows:

CSET(MUSTBRANCH (COMP)) CSET(MUSTDOM(S (VP) NP(N IT)))

and the tree below has the circled nodes as ones which are to be examined for prunability:



then those nodes pointed to by single arrows will be pruned. In addition, since the parent of each pruned node is examined, the node pointed to by the double arrow will also be pruned. The resulting tree will be:



The above conventions are adequate for the grammars we have experienced with so far, but they could be considerably generalized. Since it would be easy to modify the program in a number of different ways, it seems wisest to wait until we see what kind of conventions a number of different linguists want and then try to make up a system whereby each linguist can set a parameter to get the pruning conventions he wants.

Functions for applying transformations

SETPARS(tree) [set parents]

where 'tree' is the name of a tree. This functions should be executed before the user calls APTRANS on his own to apply a list of transformations to 'tree'. It sets a list of parent pointers which the SC functions require. If the user used the cycle functions (DOTRANS, DOCYCLE) then he does not have to use SETPARS because those functions take care of it for him.

APTRANS(trans tree) [apply transformations]

where 'trans' is a list whose elements are the names of transformations defined by DEFTRAN or the name of such a list; the 'tree' is a tree which is a (not necessarily proper) subtree of a tree which has had SETPARS(tree) performed on it or is the name of such a tree.

The transformations whose names are in the list 'trans' are applied in order listed to the tree 'tree'. As each transformation is applied, its name is printed. If the variable ONLIN has been set to T (e.g., by the user's typing

CSET(ONLIN T)

before he typed APTRANS), then a list of the nodes matching the segments of the SD and the new tree will be printed before the SC has applied.

If the transformation is optional, then when it has been determined that the SD (and restrictions, if any) are met, the program will type

tnam IS OPTIONAL. YES OR NO ----

where 'tnam' is the name of the transformation. The user may then do one of the following:

1. Type 'yes' and have the transformation apply.
2. Type 'no' and have it skipped.
3. Type 'show' and have the tree printed (this is useful if ONLIN is NIL.
4. Type 'showfrom' n, which does a PNODE(n).
5. Type 'how' and have the list of the numbers of the nodes matching the SD segments printed.
6. Type 'Q' followed by any function name and list of arguments. The function will apply to the list of arguments and the value will be printed.
7. Type 'E' and any expression. The expression will be evaluated and the value printed.

If the user exercises options 4,5,6 or 7 the program expects the user to exercise one of the seven options again. Options 6 or 7

(which are really ways of escaping back into LISP on a temporary basis) will be useful as an aid in debugging to users making changes to the program.

If the transformation is iterative and 'n' matches are found (where 'n' is greater than 1), then "n MATCHES" will be printed before any of the SC's are carried out. If the transformation is also optional, the user will be asked whether he wants to have it done (and be given the 7 options above) for each match.

If the transformation is recursive, then after all SC's have been performed, the entire procedure is applied all over again to the changed tree. However, if the "changed" tree is not, in fact, different from the tree it used to be before, then

tnam WOULD HAVE LOOPED

is printed and the transformation is not tried again.

Example of use:

APTRANS((COMPLA COMPLB) tree)

will apply the two transformation COMPLA and COMPLB to the tree 'tree'. So will the sequence:

CSET(TRANS (COMPLA COMPLB))

APTRANS(TRANS tree)

DOTRANS(TRANS tree) [do cycle of transformations]

where 'trans' is a list of transformations or the name of such a list; and 'tree' is a tree (which must have at least one "S" node in it) or the name of such a tree. This function applies SETPARS to 'tree' and then uses APTRANS to apply the transformations in 'trans' to each

subtree of 'tree' which is dominated by an 'S' node. Before the transformations are applied to a given tree dominated by 'S', they are applied to each of its proper subtrees which are dominated by 'S' nodes (i.e., this operation coreesponds to what is called the 'cycle'). Each cycle is numbered, and (CYCLE n) is printed before the nth cycle begins. The following options apply:

1. If ONLIN is not NIL, then the tree to which the nth cycle is to be applied is printed after '(CYCLE n)' is printed.

2. IF FILE is not NIL, the file whose name is FILE1 FILE2 contains for each cycle, the cycle number, the names of each of the transformations which applied, the tree after the applications (the beginning tree being put after ORIG on the file).

When the transformations have been applied to all the cycles, then

ALL CYCLES DONE

is typed and, if ONLIN is NIL, the transformed tree is printed. Note that DOTRANS and APTRANS both result in the changing of the tree indicated by 'tree'. If it is desired to have the transformations apply to a copy of the tree, leaving the original tree intact, then the following function should be used.

DOCYCLE(tree) [do cycle of listoftrans to copy of 'tree']

where 'tree' is a tree or the name of a tree. DOTRANS is used to apply the transformation cycle with the transformations whose names are mentioned in list LISTOFTRANS to a copy of the tree 'tree'. If FILE is not NIL, then after a transformation cycle has been applied by DOTRANS (or DOCYCLE, using DOTRANS), it is possible to read back in the

information filed during the operation of the cycle. This is done by the following functions.

GETRAN(arg) [get results of transformation]

where 'arg' is either an atom (e.g., ALL) indicating that all cycles are desired, or a list of cycle numbers (corresponding to cycle numbers printed by DOTRANS). This function retrieves the filed information requested about the cycles so that it will be accessible to the next three functions.

LOOKTRAN(cycle tnam apno) [display tree resulting from transformation]

where 'cycle' is the number of the cycle desired; 'tnam' is the name of the transformation the result of which it is desired by the user to look at; and 'apno' is the number of the applications of the transformation after which the user wants to look at the tree (e.g., if the user wants to see the tree after the 3rd application of the COMPLB transformation in cycle 2, he will type

LOOKTRAN(2 COMPLB 3)

If 'apno' is omitted, it is taken as 1, i.e., LOOKTRAN (2 COMPLB) gives the same result as LOOKTRAN(2 COMPLB 1).

SUMTRAN(arg) [summarize transformation application]

prints a summary of the derivation just retrieved by GETRAN. If 'arg' is NIL (or absent) then the summary does not include trees, but only states the names of the transformations in the order they applied for

each cycle retrieved by GETRAN. If 'arg' is present and not NIL, then the tree resulting from each transformation is printed after it.

The Appendix gives examples of the use of the system. It should be noted that lower case typing was done by the user and upper case typing by the program.

REFERENCES

Blair, F. Programming of the grammar tester. Specification and utilization of a transformational grammar. Scientific Report No. 1, IBM Corporation, Yorktown Heights, N. Y., 1966.

Friedman, J. Computational linguistics project reports. Stanford University, 1966-1968.

Gross, L. N. On-line programming system: user's manual. MTP-59, MITRE Corporation, 1967.

Londe, D. L., and Schoene, W. J. TGT: Transformational grammar tester. AFIPS Conference Proceedings: Spring Joint Computer Conference, 1968, 32, 385-393.

Walker, D. E., Chapin, P. G., Geis, M. L., and Gross, L. N. Recent developments in the MITRE syntactic analysis procedure. MTP-11, MITRE Corporation, 1966.

Zwicky, A. M., Friedman, J., Hall, B. C., and Walker, D. E. The MITRE syntactic analysis procedure for transformational grammars. AFIPS Conference Proceedings: Fall Joint Computer Conference, 1965, 27, 317-326.

APPENDIX

A SCRIPT WITH COMMENTS

Preliminary Comments on Person-Program Communication

In addition to communicating directly with the rule tester (the means for which is discussed in the main body of this paper), the user will have to type directly to the time-sharing monitor (hereafter simply called "the monitor") and some of the service programs it calls. The only service the monitor directly performs for the user is to give control to a program whose name the user types, and to hand that program any arguments that he types on the same line.

The user's typed input is processed either directly by the monitor or by some program that the monitor has given control to. In the latter case, the format of the input depends on the program involved (those for the rule tester have been discussed in the main body of the paper). If the monitor gets the input, then it expects the input to be the name of some program that it has, followed by possible arguments, followed by a carriage return. For example, one of the programs available to the user will print any text files. The name of the program is PRINT, and its arguments are the first and second names of the file to be printed. Thus, to print a file named ROSTRN LISP, the user should type

```
print rostrn lisp
```

The monitor responds to each command by typing

```
W time
```

where 'time' is replaced by the current time of day, and then giving control to the program whose name is the first word typed. When this program returns control to the monitor, the monitor types

R t_1+t_2

where 't_1' is the time in seconds the computer was actually paying attention to the user, and 't_2' is the time spent bringing his program in and out of the computer (the user is charged for the total time 't_1'+'t_2').

Before this session was begun, the file named LING SAVED had been made up to have the LISP programs described in the body of this paper. In addition, MUSTBRANCH had been set to (S COMP) so that any node whose name is either S or COMP will be pruned if it is checked for prune-ability and it doesn't branch (cf. p. 23).

The file ROSTRN LISP (which the session begins by printing) contains LISP functions which cause

a) a specification of default options for the transformations

b) definitions of the transformations

c) specification of the names of the transformations in the order they are to be applied.

1. The user begins by telling the monitor to call the program PRINT to print the file ROSTRN LISP. Note that the monitor responds by telling the user that the time is 3.2 minutes after midnight before calling on the printing program. The file is duly printed by the printing program, which then returns control to the monitor, which tells the user that the print program has used 1.700 seconds of processing time to work on the user's request, plus .500 seconds of time to swap the programs serving the user in and out of the computer's main memory (so that other users could have their requests processed during the same period). Note that, for the structural descriptions

```
print rostrn lisp                                    (1)
W 003.2

ROSTRN   LISP      12/30  0003.3

$$$ THIS FILE CONTAINS TRANSFORMATIONS IN THE LATEST, MOST UP TO DATE
FORMAT.  $
TRANOPT((NOPT NRECURS ITER BND))
DEFTRAN((COMPLA NITER
          (X IT S X)
          CADJL(THAT 3)
                    ))
DEFTRAN((COMPLB OPT
          (X IT ( THAT (NP VP))V)
          (' 2 S  1  S) 7 8
           SB(FOR 4)
           CADJL(TO 7)
                    ))
DEFTRAN((ITREPL
          (X ( IT (FOR (NP VP)))VP)
          ( NP 3 S  5  S) 7 8  9
           SB(-7 3) 4    6
          CADJR(-4 9)
                    ))
DEFTRAN((THERIN
          (NP (BE X))
          (  VP  ) 4
           SB(THERE 1)
          ARS(1 3)
                    ))
DEFTRAN((ITEXT  OPT
          ( (X ( IT S)X))
          ( S  NP) 4 5 6
          ARB(-5 1)
                    ))
DEFTRAN((THATDEL OPT
          (X V ( IT (THAT S))X)
          ( 2 NP  4  S) 6 7 8
          ERASN((6)) 5
                    ))
DEFTRAN((ITDEL
          (X ( IT S) X)
           ( NP) 3 4  5
          ERASN((3))
                    ))
```

in the transformation definitions the SD numbers which are implicitly assigned by the program are written in.

2. After the file LING SAVED is given control, all further input from the user (until the monitor gets control back) is processed by the rule tester and must follow the conventions described in the main text of this paper.

3. 'excise' is a LISP function which causes more room to be made available to the user by excising the LISP compiler (which is needed only if the user wants to change one of the rule tester functions). Ideally, the user should not be bothered with details of this sort, and he could regard it as a meaningless incantation which somehow helps the system to function more smoothly. The 'loadg' command (cf. p. 7) loads the file ROSTRN LISP printed above, after which the transformations in it are defined, and the order in which DOCYCLE will apply them is that specified in the last two lines of ROSTRN LISP.

```
DEFTRAN((REFL
          (X (N X)X NP X)
          ( NP2 )
          6 EQA(2)
          ARD(PRO 6)
                    ))
DEFTRAN((PRONOMA  OPT
          (X (N X) X NP X)
          ( NP2 )
          6 EQA(2)
          ARD(PRO 2)
                    ))
DEFTRAN((PRONOMB NBND
          (X (N X) X NP X)
          ( NP2 )
          6 EQA(2)
          ARD(PRO 6)
                    ))
DEFTRAN((COMPDEL
          (X ( FOR VP) X)
          ( S)
          ERASN((3))
                    ))
CSET(LISTOFTRANS(COMPLA COMPLR ITREPL THERIN ITEXT THATDEL
ITDEL REFL PRONOMA PRONOMB COMPDEL))
R 1.700+.500
```

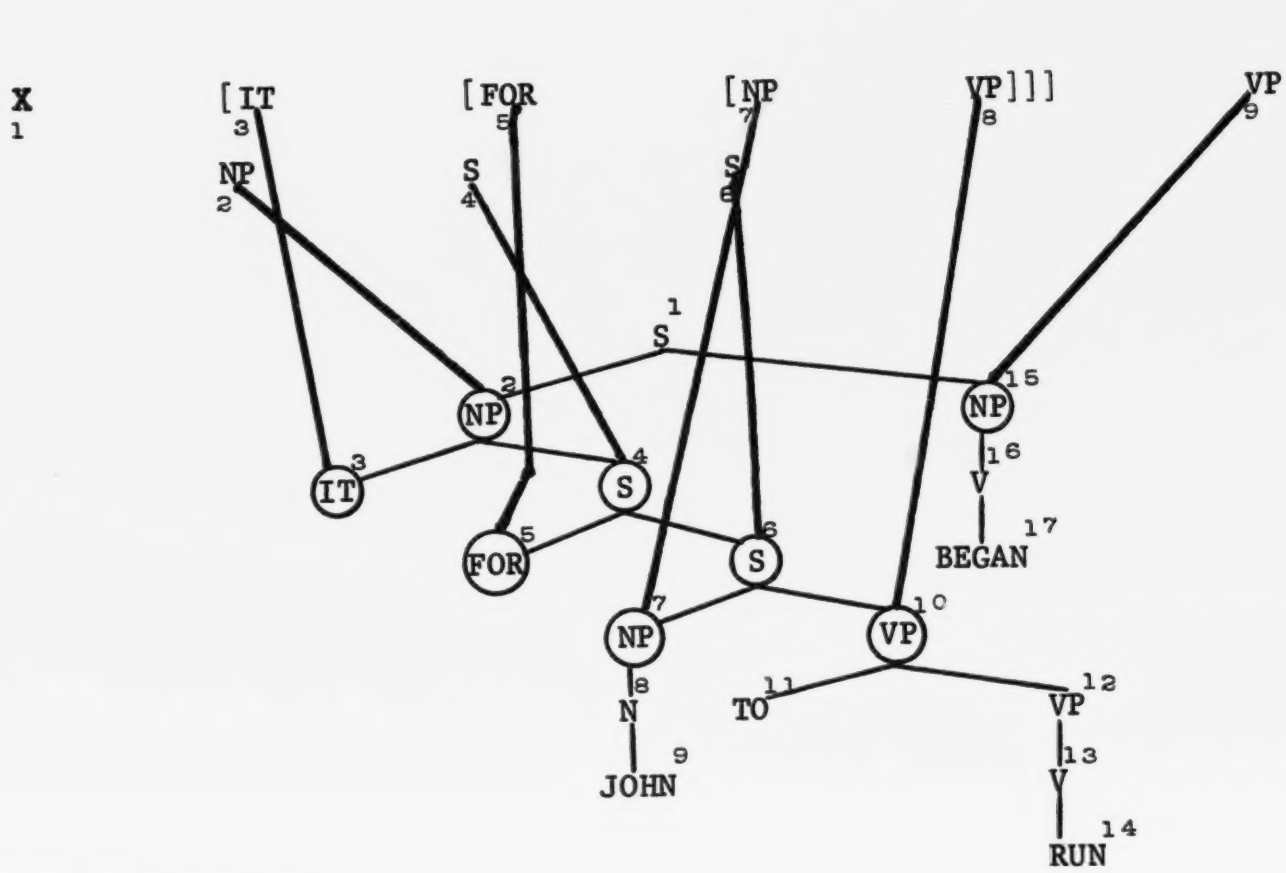


```
r ling                    (2)
W 007.5
excise(t)  loadg((rostrn))        (3)
 NIL
 ROSTRN
 NIL

cset(mplev 9)
 9
```

4. Using programs described on page 12 the user sets up a tree called JOHNRUN, which he hopes is the base tree which the transformations will transform into the surface tree for "John began to run." Note that this is the same tree that was shown on page 8.

5. After setting the variable ONLIN to T so that the tree will be printed after each transformation has applied, the function DOCYCLE is applied to the tree defined above. It may make the operations of the transformation application programs somewhat clearer if we discuss in detail the operation of one of them. The SD of the ITREPL (IT-Replacement) transformation is repeated here, with each element of the SD connected to the node which it matches in the tree;

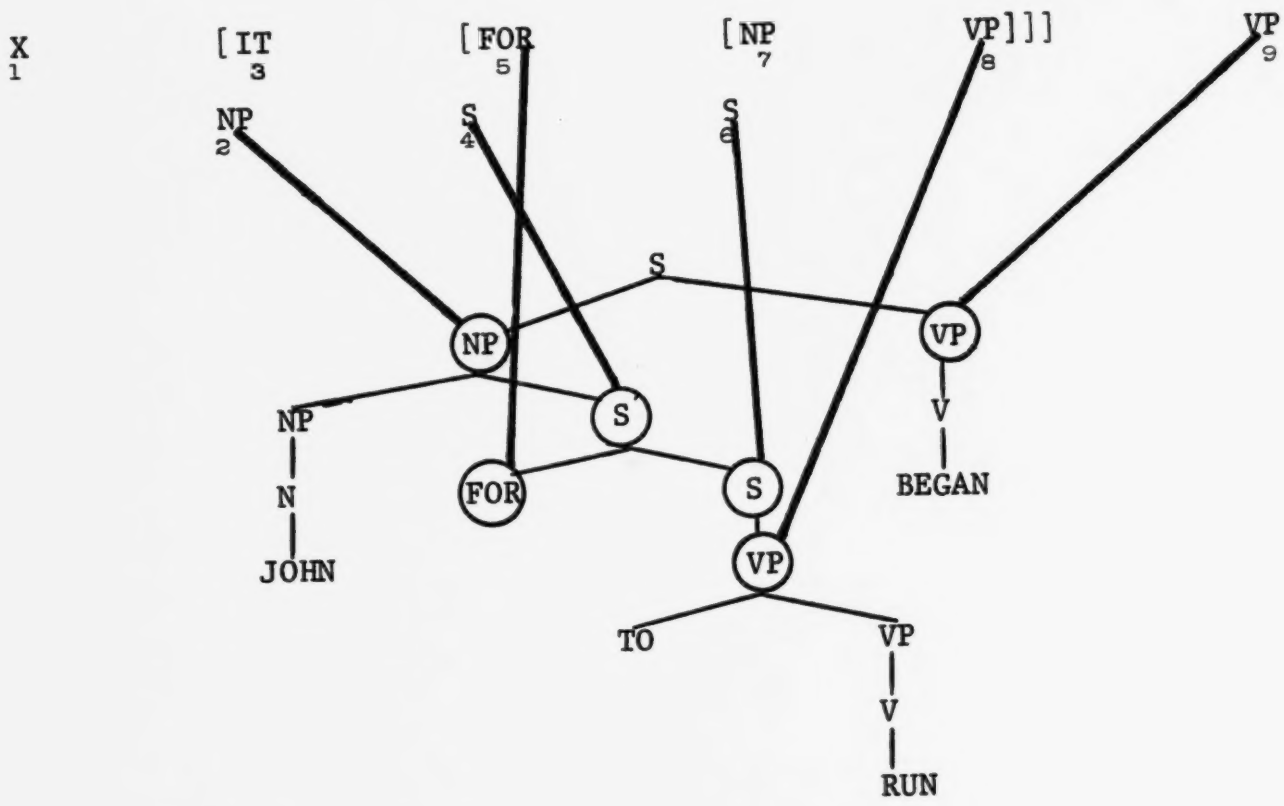


```
Intree(johnrun (s(np vp(v(began)))))                    (4)
 1S        2NP
           3VP         4V          5BEGAN
 5
chdat(np(it s(np(n(john))vp(v(eu##run)))))
 DONE
pnode(1)
 1S        2NP         3IT
                       4S          5NP         6N          7JOHN
                                   8VP         9V          10RUN
           11VP        12V         13BEGAN
cset(onlin t)
 T                                 (5)
docycle(johnrun)
 (CYCLE 1)
 4S        5NP         6N          7JOHN
           8VP         9V          10RUN
 (CYCLE 2)
 1S        2NP         3IT
                       4S          5NP         6N          7JOHN
                                   8VP         9V          10RUN
           11VP        12V         13BEGAN
 ((NIL NIL) 3 4 (11 11))
 COMPLA
 1S        2NP         3IT
                       4S          5THAT
                                   6S          7NP         8N          9JOHN
                                               10VP        11V         12RUN
            13VP       14V         15BEGAN
 ((NIL NIL) 3 4 5 6 7 10 14)
 COMPLB IS OPTIONAL.  YES OR NO ---
yes
 COMPLB
 1S        2NP         3IT
                       4S          5FOR
                                   6S          7NP         8N          9JOHN
                                               10VP        11TO
                                                           12VP        13V         14RUN
           15VP        16V         17BEGAN
 ((NIL NIL) 2 3 4 5 6 7 10 15)
 ITREPL
```

the nodes in the above tree have been numbered to correspond to the numbering in the tree that the program printed before applying ITREPL. The first structural change is

SB(-7 3)

which should take the NP whose node number is (fortuitously) 7 and substitute the entire tree dominated by it for the IT whose node number is 3, deleting the original subtree. The tree that results from this being done is

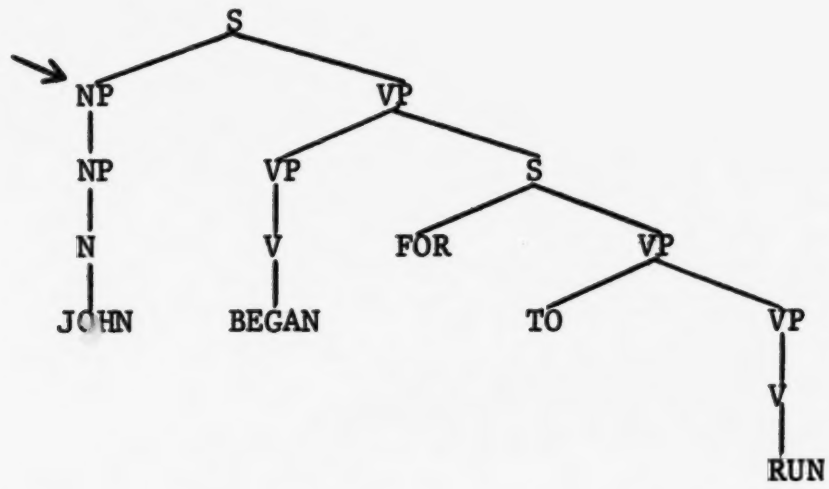


The S node corresponding to the 6th element in the SD will then be pruned since it doesn't branch and S is on the list MUSTBRANCH of symbols of nodes which are pruned if they do not branch. Those elements of the SD which now do not have lines connecting them to

nodes in the new tree no longer correspond to any part of the tree. The next structural change is

CADJR(-4 9)

which should take the entire subtree dominated by the S node connected to the 4th element in the SD and Chomsky adjoin it to the right of the VP connected to the last element of the SD, and then delete it from the position it previously occupied in the tree. Note that the tree being moved has been changed by the previous structural change in this transformation, and it is the changed subtree that is Chomsky adjoined. The resulting tree is:



The NP node pointed to by an arrow in the above tree will be pruned since it is of the form $\overset{A}{\underset{A}{|}}$. After pruning, the resulting tree is as shown in the printout below the name ITREPL in the script.

6. The tree resulting from the application of the transformations to both cycles is that shown after the last transformation. The user prints the leaves (i.e., terminal symbols) of the original base tree and then the leaves of the tree resulting from the last transformation (note that the last tree printed can be referred to as LSTPTREE). Being satisfied that the transformations acted on the base tree JOHNRUN to produce the expected surface tree, the user files the tree on the file STORAG LISP.

7. Typing STOP is one way of getting out of the LISP system and back to the time-sharing monitor. Note that the above proceedings cost 10.65 seconds of computing time, and 12.716 seconds of "swapping" time. Thus, were the same operations to be performed without the benefit of time-sharing, it would have cost less than half as much.

8. We print the file STORAG LISP created by the operation described in 6 above. Note that loading this file (by LOADG((STORAG))) will result in the symbol JOHNRUN being given the tree created above as its value.

```
1S        2NP       3N        4JOHN
          5VP       6VP       7V        8BEGAN
                    9S        10FOR
                              11VP      12TO
                                        13VP      14V       15RUN
((2 6) 9 10 11 (NIL NIL))
COMPDEL
1S        2NP       3N        4JOHN
           5VP      6VP       7V        8BEGAN
                    9VP       10TO
                              11VP      12V       13RUN
 ALL CYCLES DONE
```

```
pleafs(johnrun)
 (IT JOHN RUN BEGAN)
pleafs(lstptree)
 (JOHN BEGAN TO RUN)
prtree(johnrun)
 1S        2NP       3IT
                     4S        5NP       6N        7JOHN
                               8VP       9V        10RUN
           11VP      12V       13BEGAN
 13
filetree(johnrun storag)
 NIL
stop
R 10.650+12.716
```

⑥

⑦

```
print storag lisp
W 014.0

STORAG   LISP      12/30  0014.1

TREESET
(JOHNRUN (S (NP (IT S (NP (N (JOHN)) VP (V (RUN)))) VP (V (BEGAN)))))
R .950+.283
```

⑧

9. The user starts again, but by reading in the file STORAG LISP he may begin with the tree JOHNRUN already defined.

10. The first two uses of 'intree' serve to create "skeletons" of trees to be used below in constructing the tree MAMMOTH. Note that the tree MAMMOTH is begun with a copy of the skeleton SSUBJ. Note that the tree JOHNRUN read in from STORAG LISP is used as the most deeply embedded S. The base structure built is the one which the user thinks underlies the sentence "Mary seems to want Melvin to expect her to know that John began to run."

(9)

```
r ling
W 015.0
c@ excise(t)  loadg((rostrn storag))
 NIL
 ROSTRN
 STORAG
 NIL
intree(ssubj (s(np(it ss)vp(vv))))                    (10)
 1S        2NP        3IT
                      4SS
           5VP        6VV
 6
intree(sobj(s(np(nn)vp(vv np(it ss)))))
 1S        2NP        3NN
           4VP        5VV
                      6NP        7IT
                                 8SS
 8

intree(mammoth ssubj)
 1S        2NP        3IT
                      4SS
            5VP       6VV
 6
chnode(vv(v(seems)))  chnode(4 sobj)  pnode(1)
 DONE
 DONE
 1S        2NP        3IT
                      4S         5NP        6NN
                                 7VP        8VV
                                            9NP        10IT
                                                       11SS
            12VP      13V        14SEEMS


chnode(8(v(want))) chnode(6(n(mary)))  chnode(ss sobj)
 DONE
 DONE
 DONE
pnode(11)
```

(This page--with the exception of this sentence, its parentheses, and the page number--is intentionally blank.)

```
 11NP      12IT
           13S       14NP      15NN
                     16VP      17VV
                               18NP      19IT
                                         20SS

chnode(17 (v(expect))) chnode(15(n(melvin))) chnode(ss(# sobj) pnode(15)
 DONE
 DONE
 DONE
 15N       16MELVIN
pnode(17)
 17VP      18V       19EXPECT
           20NP      21IT
                     22S       23NP      24N"
                               25VP      26VV
                                         27NP      28IT
                                                   29SS
chnode(nn 6) chnode(vv(v(know))) chnode(ss johnrun) pnode(1)
 DONE
 DONE
 DONE
 DONE
 DONE
 1S        2NP       3IT
                     4S        5NP       6N        7MARY
                               8VP       9V        10WANT
                                         11NP      12IT
                                                   13S       14NP      15N       16MELVIN
                                                             17VP      18V       19EXPECT
                                                                       20NP      21IT
                                                                                 22S       23NP      24N ..
                                                                                           26VP      27V ..
                                                                                                     29NP .
           44VP      45V       46SEEMS

pnode(22)
 22S       23NP      24N       25MARY
           26VP      27V       28KNOW
                     29NP      30IT
                               31S       32NP      33IT
                                                   34S       35NP      36N       37JOHN
                                                             38VP      39V       40RUN
                                         41VP      42V       43BEGAN

pleafs(mammoth)
 (IT MARY WANT IT MELVIN EXPECT IT MARY KNOW IT IT JOHN RUN
 BEGAN SEEMS)
```

11. This time, in trying the transformations, we do not have the tree printed after each transformation. The first two cycles are identical with the derivation of "John began to run," so it is not necessary for the user to look at the tree for his first choice, since he knows from before that he will want COMPLB to apply. However, in cycle 3, the user is not sure how ITEXT (It-Extraposition) would apply, so he asks how, is given a list of node numbers (which do not correspond to the numbers in the print of MAMMOTH since changes have occured in the copy being worked on), and asks to see the tree to which the transformation will apply. In this case, it would take the subtree dominated by the node numbered 31 ("that John began to swim") and place it as the rightmost duaghter of the node numbered 22 (hence, also as the right sister of the node numbered 26). This will result in a change in structure, but not in the terminal string. The user decides not to execute ITEXT here, and the next transformation that will apply is THAT Deletion, which is also optional. The user asks how it will apply and, since the tree has not been changed since it was last printed, he does not have to print it again to see that, were THATDEL to apply it would delete the THAT whose node number is 32. Essentially, the choice is between the sentences "Mary knows John began to swim" and "Mary knows that John began to swim." The user then decides not to apply the THATDEL transformation here. The program then finds ITDEL (IT-Deletion) is the next applicable transformation this cycle, and deletes the IT whose node is numbered 30.

12. On the next cycle, (CYCLE 4), COMPLA applies and then the next applicable transformation is ITEXT. This means that COMPLB was

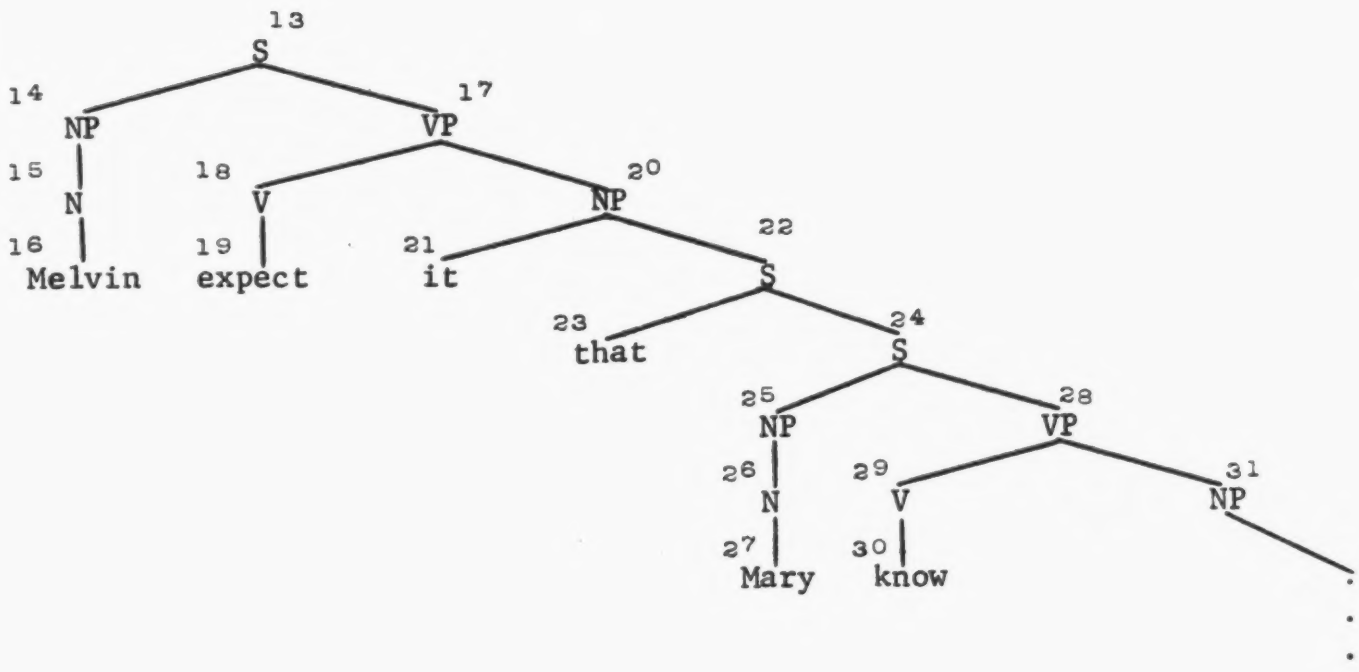
```
docycle(mammoth)
 (CYCLE 1)
 (CYCLE 2)
 COMPLA
 COMPLB IS OPTIONAL.  YES OR NO ---
yes
 COMPLB
 ITREPL
 COMPDEL
 (CYCLE 3)
 COMPLA
 ITEXT IS OPTIONAL.  YES OR NO ---
how
 (22 (23 27) 29 30 31 (NIL NIL))
show
 22S        23NP       24N        25MARY
            26VP       27V        28KNOW
                       29NP       30IT
                                  31S        32THAT
                                             33S        34NP       35N        36JOHN
                                                        37VP       38VP       39V        40BEGAN
                                                                   41VP       42TO
                                                                              43VP       44V        45RUN
```



```
no
 THATDEL IS OPTIONAL.  YES OR NO ---
how
 ((23 23) 27 29 30 31 32 33 (NIL NIL))
no
 ITDEL
 (CYCLE 4)
 COMPLA
 ITEXT IS OPTIONAL.  YES OR NO ---
how
 (13 (14 18) 20 21 22 (NIL NIL))
showfrom 20
 20NP       21IT
            22S        23THAT
                       24S        25NP       26N        27MARY
                                  28VP       29V        30KNOW
                                             31NP       32S        33THAT
                                                                   34S        35NP       36N        37JOHN
                                                                              38VP       39VP       40V        41BEGAN
                                                                                         42VP       43TO
                                                                                                    44VP       45V ...

 q #INT. IGNORED.
 QUIT,
P .000+.000
```

not applicable. However, the user had expected it to apply, so as to get "Melvin expected it for Mary to know that John began to run." The structure against which the SD of COMPLB was tested is



We see from this that the tree could not meet the SD of COMPLB because COMPLB requires a V at the end after the VP numbered (28). However, we would like COMPLB to apply, so as to allow the derivation of "Melvin expects Mary to know ...". The user at this point decides to change the transformation COMPLB so that the final V is optional. To change the transformation, he must go back to the text of its definition on the file ROSTRN LISP. He wishes to preserve what he has already done so that, after changing the transformation on the file ROSTRN LISP and reading in the new version of that transformation, he can resume processing. The user can return control to the time-sharing system by pushing twice during a short period of time the button on his console marked INTERUPT, ATTENTION, or something of that sort. This will return

control to the time-sharing monitor (usually saying QUIT followed by an indication of the time used). The reader is advised not to become perturbed at the unusual dialogue shown actually to have occurred here--there is an explanation for it, but it is much more complicated than interesting. The basic point to remember is, "keep pushing the attention button until the time-sharing monitor comes back."

13. The command shown here instructs the system to save an image of the program in use during the last command (in this case the rule tester) in the state it was when control was returned to the monitor. The first name of the new file is the first argument of the command (TEMP in this case), the last name of the new file is always SAVED. The second argument, 't', tells the save command to create the file in temporary mode (so that the number of tracks used for it won't count against the mixiumum the user is allowed.) The next command asks the system to list all files whose first names are TEMP.

14. Having saved the program, the user is ready to change the transformation COMPLB. The system provides a program for editing a text, and the command shown here causes the system to call on the editor and tell it to edit the file ROSTRN LISP. The editor, which is not described here, accepts commands for finding lines which contain certain strings of characters. Thus, the user has 'locate'd by its name the first line of the transformation to be changed, skipped down one more line to the first line of the structural description, and changed the line so that the final 'v' on it is '($ v)'. After this change has been accomplished (causing the SD of COMPLB to end optionally with a 'v' instead of obligatorily), the edited file is 'file'd back, causing a new ROSTRN LISP to replace the old one.

15. The editor is used here to print a copy of those lines of the file ROSTRN LISP which contain the definition of the revised COMPLB transformation.

```
save temp t                                    (13)
W 029.5
R 4.016+.666


edl rostrn lisp                                (14)
W 029.7
Edit
locate@locate comp1b
DEFTRAN((COMPLB OPT
bext
Not a request:
BEXT
next
print
                (X IT ( THAT (NP VP))V)
change /v)/($ v))/
                (X IT ( THAT (NP VP))($ V))

file
*
R 2.033+1.950


edl rostrn lisp                                (15)
W 033.9
Edit
locate comple#b
DEFTRAN((COMPLB OPT
print 6
DEFTRAN((COMPLB OPT
              (X IT ( THAT (NP VP))($ V))
              (     S      S )
              SB(FOR 4)
              CADJL(TO 7)
                          ))
quit
*
R 1.033+.366
```

16. The file which had been saved is resumed. The state of the file is exactly the same it had been in when it was left in 12. The first instruction given to the program is to define, from the file just changed, the transformation COMPLB which had been on it. Thus, the state of the program now differs only in that COMPLB no longer requires the 'v' at the end of the SD.

17. Having brought in the new definition of COMPLB, we are at the same point we were at the beginning of 11, except for the new COMPLB, which has been redefined so as to be applicable to trees to which it was not applicable before. Therefore, we make the same decisions that we made before until we get to the point in (CYCLE 4) where the transformation COMPLB disappointed the user by not applying. In getting to this point, COMPLB was seen to be applicable on (CYCLE 3) (allowing the generation of "... Mary knows John to began to run.", which really should be out only because of a feature on the verb). This is, of course, because the SD was changed. The user, in order to get into (CYCLE 4) with the same tree as before, decides not to apply COMPLB here.

18. This is the place at which COMPLB did not apply before its SD was changed. The user verifies that it will apply as he expects, and then tells the system to apply it. Note that it prints out the name of the transformation as "CK-LLB" instead of "COMPLB." This is almost certainly due to a glitch in the typewriter or in the data transmission between the computer and the typewriter and can be ignored.

(16)

```
r temp
W 036.4
deffile(rostrn (complb))
 NIL
docycle(mammoth)
 (CYCLE 1)
 (CYCLE 2)
 COMPLA
 COMPLB IS OPTIONAL.  YES OR NO ---
yes
 COMPLB
 ITREPL
 COMPDEL
 (CYCLE 3)
 COMPLA
 COMPLB IS OPTIONAL.  YES OR NO ---
no
 ITEXT IS OPTIONAL.  YES OR NO ---
no
 THATDEL IS OPTIONAL.  YES OR NO ---
no
 ITDEL
 (CYCLE 4)
 COMPLA
 COMPLB IS OPTIONAL.  YES OR NO ---
how
 ((14 18) 21 22 23 24 25 28 NIL)
showfrom 22
 22S      23THAT
          24S       25NP      26N       27MARY
                    28VP      29V       30KNOW
                              31NP      32S       33THAT
                                                  34S       35NP      36N       37JOHN
                                                            38VP      39VP      40V       41BEGAN
                                                                      42VP      43TO
                                                                                44VP      45V       46RUN

yes
 CK_LLB
 ITEXT IS OPTIONAL.  YES OR NO ---
```

(17)

(18)

19. The user had expected ITREPL to apply here, producing "... Melvin expects Mary to know that ...". The fact that he is being asked about ITEXT (which comes later in the list of transformations LISTOFTRANS) tells him that it was not applicable as defined. The problem essentially is that the transformation expects a VP node to follow the S node numbered 22 in the tree shown. In fact, the transformation expects to Chomsky Adjoin that S to that VP. If there were no such node (i.e. if the last VP in the SD were made optional and matched NIL) then the CADJR change would simply not apply, but it would still be possible to replace the IT numbered 21 with the NP numbered 25. The user decides to change ITREPL so as to allow this.

20. After returning to the monitor and saving the current state of the rule tester, the user edits the file containing the transformation ITREPL, changing ITREPL so as to make the final VP of the SD optional. Since the user feels that the changed ITREPL may be applicable to some trees to which he would not want it applied, he makes it an optional transformation so that he will be asked whether to apply it before each time.

(19)

```
show how
 13S        14NP       15N        16MELVIN
            17VP       18V        19EXPECT
                       20NP       21IT
                                  22S        23FOR
                                             24S        25NP       26N        27MARY
                                                        28VP       29TO
                                                                   30VP       31V        32KNOW
                                                                              33NP       34S        35THAT
                                                                                                    36S        37NP ...
                                                                                                               40VP ...

 (13 (14 18) 20 21 22 (NIL NIL))
INT. 0
***  ERROR  CALLED
 -
 QUIT,
R 9.033+14.533
```


(20)

```
save temp t
W 041.8
R 4.083+.683

edl rostrn lisp
W 042.0
Edit
locate itrepl
DEFTRAN((ITREPL
append opt
print
DEFTRAN((ITREPLOPT
change /lo/l o/
DEFTRAN((ITREPL OPT
next
print
          (X ( IT (FOR (NP VP)))VP)
change )#change /)vp/)(change /)vp/)($ vp))#/
          (X ( IT (FOR (NP VP)))($ VP))
top
locate itrepl
DEFTRAN((ITREPL OPT
```

21. This is the new definition of ITREPL.

22. The user resumed the rule tester as it was when he saved it, reads in the new definition of ITREPL, and goes through a number of derivations. This last section also demonstrates the use of the functions GETRAN, SUMTRAN, and LOOKTRAN for getting information about the most recently run derivation after it has been run.

(21)

```
print 6
DEFTRAN((ITREPL OPT
          (X ( IT (FOR (NP VP)))($ VP))
          ( NP   S    S)
          SB(-7 3)
          CADJR(-4 9)
                 ))
file
*
R 2.050+2.016
```

(22)

```
r temp
W 045.4
deffile(rostrn (itrepl))   docycle(mammoth)
 NIL
 (CYCLE 1)
 (CYCLE 2)
 COMPLA
 COMPLB IS OPTIONAL.  YES OR NO ---
yes
 COMPLB
 ITREPL IS OPTIONAL.  YES OR NO ---
yet
  I DIDN'T UNDERSTAND THAT.
yes
 ITREPL
 COMPDEL
 (CYCLE 3)
 COMPLA
 COMPLB IS OPTIONAL.  YES OR NO ---
how
 ((23 27) 30 31 32 33 34 37 NIL)
show
 22S       23NP      24N       25MARY
            26VP      27V       28KNOW
                      29NP      30IT
                                31S       32THAT
                                          33S       34NP      35N       36JOHN
                                                    37VP      38VP      39V       40BEGAN
                                                              41VP      42TO
                                                                        43VP      44V       45RUN
```

```
no
 ITEXT IS OPTIONAL.  YES OR NO ---
how
 (22 (23 27) 29 30 31 (NIL NIL))
no
 THATDEL IS OPTIONAL.  YES OR NO ---
how
 ((23 23) 27 29 30 31 32 33 (NIL NIL))
no
 ITDEL
 (CYCLE 4)
 COMPLA
 COMPLB IS OPTIONAL.  YES OR NO ---
yes
 COMPLB
 ITREPL IS OPTIONAL.  YES OR NO ---
how
 ((14 18) 20 21 22 23 24 25 28 NIL)
showform@ showfrom 20
 20NP     21IT
          22S       23FOR
                    24S       25NP      26N       27MARY
                              28VP      29TO
                                        30VP      31V       32KNOW
                                                  33NP      34S       35THAT
                                                                      36S       37NP      38N       39JOHN
                                                                                40VP      41VP      42V ...
                                                                                          44VP      45TO
                                                                                                    46VP ...

yes
 ITREPL
 COMPDEL
 (CYCLE 5)
 COMPLA
 COMPLB IS OPTIONAL.  YES OR NO ---
```

```
show how
 4S          5NP          6N          7MARY
              8VP          9V          10WANT
                           11NP        12IT
                                       13S          14THAT
                                                    15S          16NP          17N          18MELVIN
                                                                 19VP          20V          21EXPECT
                                                                               22NP         23NP          24N          25MARY
                                                                                            26VP          27TO
                                                                                                          28VP         29V          30KNOW
                                                                                                                       31NP         32S ...

  ((5 9) 12 13 14 15 16 19 NIL)
 yes
  COMPLB
  ITREPL IS OPTIONAL.  YES OR NO ---
 how
 ((5 9) 11 12 13 14 15 16 19 NIL)
 showfrom 12
  12IT
  13S          14FOR
               15S          16NP          17N          18MELVIN
                            19VP          20TO
                                          21VP         22V          23EXPECT
                                                       24NP         25NP          26N          27MARY
                                                                    28VP          29TO
                                                                                  30VP         31V          32KNOW
                                                                                               33NP         34S          35THAT
                                                                                                                         36S          37NP ...
                                                                                                                                      40VP ...

 yes
  ITREPL
  PRONOMB
  COMPDEL
  (CYCLE 6)
  COMPLA
  COMPLB IS OPTIONAL.  YES OR NO ---
```

```
how
 ((NIL NIL) 3 4 5 6 7 10 49)
showfrom 49
 49V       50SEEMS
yes
 COMPLB
 ITREPL IS OPTIONAL.  YES OR NO ---
yes
 ITREPL
 COMPDEL
  ALL CYCLES DONE
 1S        2NP       3N        4MARY
           5VP       6VP       7V        8SEEMS
                     9VP       10TO
                               11VP      12V       13WANT
                                         14NP      15NP      16N       17MELVIN
                                                   18VP      19TO
                                                             20VP      21V       22EXPECT
                                                                       23NP      24NP      25N       26MARY
                                                                                           27PRO
                                                                                 28VP      29TO
                                                                                           30VP      31V ...
                                                                                                     33NP ...

 48
pnode(30)
 30VP      31V       32KNOW
           33NP      34S       35THAT
                               36S       37NP      38N       39JOHN
                                         40VP      41VP      42V       43BEGAN
                                                   44VP      45TO
                                                             46VP      47V       48RUN

pleafs(1stptree)
 (MARY SEEMS TO WANT MELVIN TO EXPECT MARY PRO TO KNOW THAT
 JOHN BEGAN TO RUN)

pleafs(am##mammoth)
 (IT MARY WANT IT MELVIN EXPECT IT MARY KNOW IT IT JOHN RUN
 BEGAN SEEMS)
getran(all)
  USE LOOKTRAN .
```

```
sumtran()
 (CYCLE 0)
 (TREE)
 (CYCLE 1)
 (ORIG)
 (CYCLE 2)
 (ORIG COMPLA COMPLB ITREPL COMPDEL)
 (CYCLE 3)
 (ORIG COMPLA ITDEL)
 (CYCLE 4)
 (ORIG COMPLA COMPLB ITREPL COMPDEL)
 (CYCLE 5)
 (ORIG COMPLA COMPLB ITREPL PRONOMB COMPDEL)
 (CYCLE 6)
 (ORIG COMPLA COMPLB ITREPL COMPDEL)
 NIL

docycle(mammoth)
 (CYCLE 1)
 (CYCLE 2)
 COMPLA
 COMPLB IS OPTIONAL.  YES OR NO ---
yes
 COMPLB
 ITREPL IS OPTIONAL.  YES OR NO ---
yes
 ITREPL
 COMPDEL
 (CYCLE 3)
 COMPLA
 COMPLB IS OPTIONAL.  YES OR NO ---
no
 ITEXT IS OPTIONAL.  YES OR NO ---
show how
 22S      23NP     24N      25MARY
          26VP     27V      28KNOW
                   29NP     30IT
                            31S      32THAT
                                     33S      34NP     35N      36JOHN
                                              37VP     38VP     39V      40BEGAN
                                                       41VP     42TO
                                                                43VP     44V      45RUN

 (22 (23 27) 29 30 31 (NIL NIL))
no
 THATDEL IS OPTIONAL.  YES OR NO ---
```

```
how
 ((23 23) 27 29 30 31 32 33 (NIL NIL))
yes
 THATDEL
 ITDEL
 (CYCLE 4)
 COMPLA
 COMPLB IS OPTIONAL.  YES OR NO ---
q pleafs(lstptree)
 (IT MARY WANT IT MELVIN EXPECT IT THAT MARY KNOW JOHN BEGAN
 TO RUN SEEMS)
yes
 COMPLB
 ITREPL IS OPTIONAL.  YES OR NO ---
yes

 ITREPL
 COMPDEL
 (CYCLE 5)
 COMPLA
 COMPLB IS OPTIONAL.  YES OR NO ---
yes
 COMPLB
 ITREPL IS OPTIONAL.  YES OR NO ---
yes
 ITREPL
 PRONOMB
 COMPDEL
 (CYCLE 6)
 COMPLA
 COMPLB IS OPTIONAL.  YES OR NO ---
no
 ITEXT IS OPTIONAL.  YES OR NO ---
how
 (1 (NIL NIL) 2 3 4 (46 46))
yes
 ITEXT
  ALL CYCLES DONE
```

```
1S        2NP       3IT
           4VP       5V        6SEEMS
           7S        8THAT
                     9S        10NP      11N       12MARY
                               13VP      14V       15WANT
                                         16NP      17NP      18N       19MELVIN
                                                   20VP      21TO
                                                             22VP      23V       24EXPECT
                                                                       25NP      26NP      27N       28MARY
                                                                                           29PRO
                                                                                 30VP      31TO
                                                                                           32VP      33V ...
                                                                                                     35NP ...

 48
pleafs(1stptree)
 (IT SEEMS THAT MARY WANT MELVIN TO EXPECT MARY PRO TO KNOW
 JOHN BEGAN TO RUN)
getran((5 6))
  USE LOOKTRAN .
sumtran()
 (CYCLE 5)
 (ORIG COMPLA COMPLB ITREPL PRONOMB COMPDEL)
 (CYCLE 6)
 (ORIG COMPLA ITEXT)
 NIL
looktran(5 pronomb)
 1S        2NP       3N        4MARY
            5VP       6V        7WANT
                      8NP       9NP       10N       11MELVIN
                                12S       13FOR
                                          14VP      15TO
                                                    16VP      17V       18EXPECT
                                                              19NP      20NP      21N       22MARY
                                                                                  23PRO
                                                                        24VP      25TO
                                                                                  26VP      27V       28KNOW
                                                                                            29NP      30S ...

 42
pnode(30)
 30S       31NP      32N       33JOHN
            34VP      35VP      36V       37BEGAN
                      38VP      39TO
                                40VP      41V       42RUN

 QUIT,
R 50.700+40.>50
```